JPL: CL#02-2877

Mission Data System

Framework Description

December 15, 2005
Version 2.5

Approved by:

Kenny Meyer , MDS Project Manager	Date

Jet Propulsion Laboratory
California Institute of Technology

Mission Data System

Framework Description

This document provides an overall description of the MDS Framework technology. Since the purpose is to provide a general reference for the frameworks, the descriptions are organized as compendium. This document does not provide guidance for how the MDS technology should be used.

This document includes 3 Sections:

- Section 1 provides a summary of the framework and a brief description of each framework package.
- Section 2 provides a description for each individual package including package functionality and relevant glossary items.
- Section 3 provides an index of glossary terms and abbreviations.

NOTE: The frameworks are intended to describe functionality that is independent of a specific programming language. Nonetheless, this document includes many references to C++ constructs because the current MDS implementation is in C++. Implementations have been built in the Java programming language as well. References to C++ constructs have been included here because they address C++ specific issues, or because they describe capabilities that have not, as yet, been generalized.

1 Overview of Framework Packages and Layers

The MDS-based systems are built from a reusable set of core capabilities. The complete set of these core capabilities is called a framework.

The framework is organized into a set of packages. Each package contains a set of related functions created to satisfy a capability area. The packages are described in detail in Section 2, "Frameworks."

The packages are organized into layers. These layers are used to manage the dependencies between code elements. Dependencies only flow down from higher layers to lower layers. For example, level-1 packages may *not* depend on packages in levels 2, 3, 4 or 5. Level-3 packages may depend on packages in level 1, 2 or 3, but may *not* depend on packages in levels 4, or 5. And so on.

The levels are organized roughly into types of functionality. This is, at best, a labeling of convenience and should not be seen as a rigorous organizing principle.

Level 0: Operating System Services	Externally provided standard libraries and interfaces needed to communicate with target operating systems.
Level 1: Primitive Services	Generic programming capabilities that are commonly required for the development of embedded systems.
Level 2: Simple Services	Low-level services needed for debugging, runtime configuration and temporal programming
Level 3: Complex Services	Generic database capabilities
Level 4: State Services	Support services of the MDS state-based paradigm
Level 5:Application Services	Capabilities needed for runtime construction, system operations, system test, performance monitoring, and generic visualization tools

The layer diagram in Figure 1 depicts six layers in the MDS core framework and the packages in each layer. Each package is briefly described in Table 1, “Brief description of MDS Framework Packages .”

Figure 1: The MDS Framework is layered

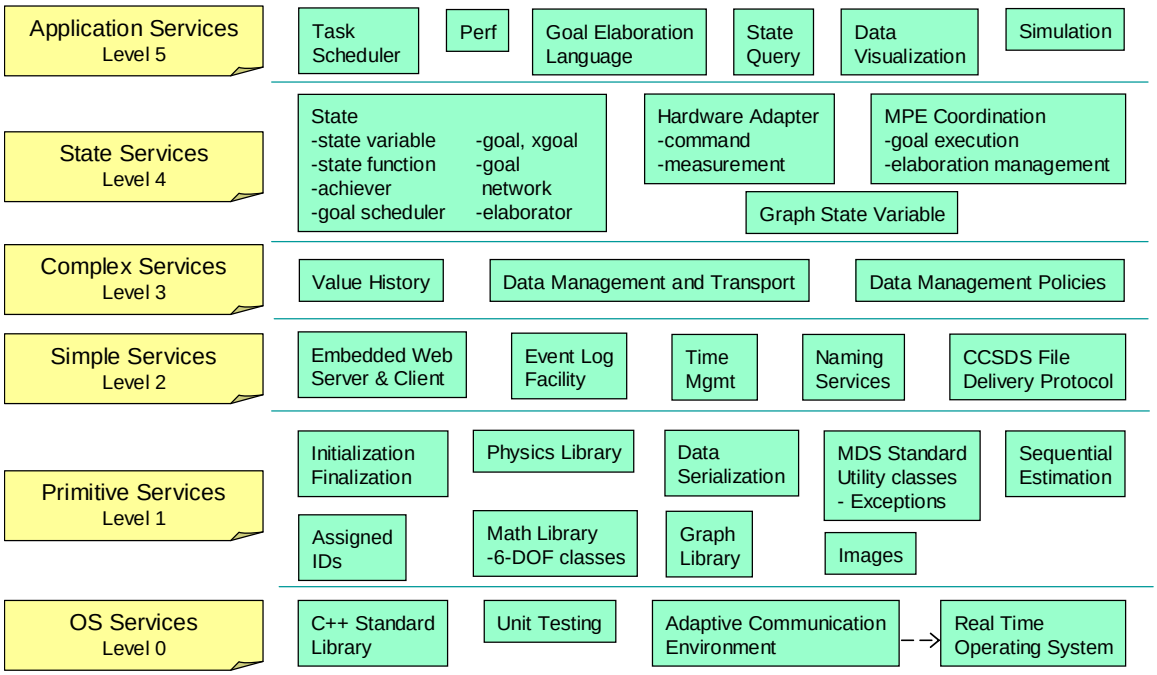

Table 1: Brief description of MDS Framework Packages

	Package	Description
1.	ACE	Provides platform independent operating system primitives for core patterns for concurrent communication software. (ACE is an acronym for Adaptive Communication Environment).
2.	Assigned Ids	Provides an interface for importing externally assigned identifiers into a deployment.
3.	CCSDS File Delivery Protocol	Provides telecommunications protocols for reliable transport of data products over space links. (CCSDS is an acronym of, which spelled out is Consultative Committee for Space Data Systems).
4.	C++ Standard Library	Provides C++ language library functions.
5.	Data Management and Transport	Organizes data storage and provides query mechanisms. Provides abstract interfaces that must be supported by a MDS-compliant transport service.
6.	Data Management Policies	Provides a set of primitives for controlling data management and transport.
7.	Data Serialization	Provides interfaces for encoding and decoding data for efficient and portable storage, retrieval, and transmission.
8.	Data Visualization	Provides graphical displays of state queries.
9.	Embedded Web Server and Client	Provides a thin web server following the HTTP 1.0 protocol. Provides a thin web client following the HTTP 1.0 protocol. Enables interaction with an embedded web server via the HTTP protocol, in the same way that browsers talk to web servers. Provides a lookup service for storing and retrieving deployment addresses.
10.	Event Log Facility	Provides a general logging mechanism for reporting noteworthy events during mission operation. Provides message filtering and transport interface.
11.	Goal Elaboration Language	Provides a language for specifying the elaboration of goals and other related high-level functions.
12.	Graph	Provides the elements and algorithms need to build and use graph data structures.
13.	Graph State Variable	Provides a mechanism for defining relative states as pair-wise directed relationships between frames in a graph.
14.	Hardware Adapter	Provides base classes for commands and measurements.

	Package	Description
15.	Images	Provides base classes for images.
16.	Initialization and Finalization	Provides a mechanism for ordering the initialization and finalization objects.
17.	Math	Provides common mathematical computations for MDS frameworks, adaptations, and deployments, such as 6 Degree of Freedom relationships.
18.	MDS Standard Utility Classes	Provides utility functions that are used frequently throughout the MDS frameworks.
19.	MPE Coordination	Provides a goal execution engine and coordinates elaboration and goal checking as part of the Mission Planning and Execution (MPE) function.
20.	Naming Services	Provides capability for storing values with an associated name in a global registry.
21.	Performance	Provides tools to gather and analyze performance metrics.
22.	Physics	Provides a set of common physics objects and computations.
23.	Sequential Estimation	Provides an application-programming interface (API) for developing extended Kalman filters.
24.	Simulation	Provides elements for building simulations of systems under control.
25.	State	Provides base classes to build state variables. Provides goal-driven executable components that strive to achieve executable goals. Provides mechanisms used to construct, elaborate, and schedule temporal constraint and goal networks.
26.	State Query	Provides the functionality to query externally state and measurement histories.
27.	Task Scheduler	Provides mechanisms for building multi-threaded software architectures that can have rate-groups and periodic execution.
28.	Time Management	Provides time representations and mechanisms for manipulating them. Provides mechanisms for initialization and management of time services.
29.	Unit Testing	Provides a test harness for verifying partial adaptations of the frameworks.
30.	Value History	Provides abstract interfaces for time-tagged data containers that hold state, command, and measurement histories.

2 Frameworks

Each MDS Framework package provides a set of capabilities. This section includes an overview and a list of functions and descriptions of those functions for each framework package.

The following table provides a convenient list of the packages descriptions with page numbers.

Framework Package 1: ADAPTIVE Communication Environment (ACE)

Overview:

from ACE web-site (http://www.cs.wustl.edu/~schmidt/ACE.html).

"The ADAPTIVE Communication Environment (ACE) is a freely available, open-source object-oriented (OO) framework that implements many core patterns for concurrent communication software. ACE provides a rich set of reusable C++ wrapper facades and framework components that perform common communication software tasks across a range of OS platforms. The communication software tasks provided by ACE include event demultiplexing and event handler dispatching, signal handling, service initialization, interprocess communication, shared memory management, message routing, dynamic (re)configuration of distributed services, concurrent execution and synchronization."

"ACE is targeted for developers of high-performance and real-time communication services and applications. It simplifies the development of OO network applications and services that utilize interprocess communication, event demultiplexing, explicit dynamic linking, and concurrency. In addition, ACE automates system configuration and reconfiguration by dynamically linking services into applications at run-time and executing these services in one or more processes or threads."

"ACE is supported commercially by multiple companies using an open-source business model. In addition, many members of the ACE development team are currently working on building The ACE ORB (TAO)."

Description of basic functions:

see ACE web-site (http://www.cs.wustl.edu/~schmidt/ACE.html).

Framework Package 2: Assigned IDs

Overview:

This package provides an interface for importing externally assigned identifiers into a deployment. These Ids can be used to establish common names across deployments.

Since the C++ runtime type system doesn't try to provide a portable identifier (other than an inefficient class name string), and since we need identifiers that would be portable between languages anyway, a separate mechanism must be provided for assigning these portable IDs.

In order to align different implementations of the same class, or to correlate implementations with requirements it is much easier to manage the assignment database external to a C++ deployment, and generate the C++ or java code needed to establish the runtime associations from the assignment database.

It is assumed that an external database such as the MDS state database can be used to manage the set of identifiers, and that the database can then export a tabulation of the assignments in a form that this interface can convert into a header file for consumption in the source code.

Description of basic functions:

Function Name	Description
Create assigned IDs	Converts externally supplied IDs in the form of CSV files into enums useable in adaptation code.

Glossary:

Term	Definition
Assigned ID	Names for objects and classes that are useable across deployments.

Framework Package 3: CCSDS File Delivery Protocol (CFDP)

Overview:

The CFDP package provides an implementation of standard-compliant, data-link protocols for the reliable transport of data products between different systems over a telecommunications network. It includes methods for sending products and receiving products, and for controlling some operational parameters. The CFDP standard addresses data transport for both space and terrestrial data networks.

CFDP is an acronym of an acronym that abbreviates Consultative Committee for Space Data Systems File Delivery Protocol. That is, the primary protocol is the File Delivery Protocol (FDP), and an international standards body, the Consultative Committee for Space Data Systems (CCSDS), publishes its specifications. CCSDS communications protocols are analogous to the Internet TCP/IP protocol suite, and in fact are in some cases extensions of TCP/IP designed to work over radio links.

The file delivery protocol provides a standard method for delivering data products reliably across a link even when the underlying transport medium is unreliable. For example, a radio transmission can be interrupted by weather or misalignment of antennas or a variety of other conditions and result in products that are only partly reconstructed on the receiving side. The file delivery protocol provides an accounting mechanism on the receiving side that can detect these missing pieces and then request that those pieces be resent. The sender side of the system contains functionality to keep track of the pieces long enough to handle any subsequent requests for retransmission. Eventually, when all the holes have been filled, the receiver ends up with a complete product.

The CCSDS standards start at the radio level, and work up to the level where the received and transmitted data streams are handed off to the application software. CCSDS compliant software implements protocol that only deal with how the data bits are moved from place to place, and *not* with what these data bits are, how they are created or what they do.

Note: this is a package that is not ported to MDS 2005.

Description of basic functions:

Function Name	Description
Send product	Request that the given data product (file) be sent across an associated space link using the CFDP protocol.
Receive product	This method is a callback. When a product has been received and completely reconstructed, this method will be called to handle the product and decide what to do with it.
Control CFDP	This method encapsulates various control functions such as cancel transaction for product and recover storage.

Glossary:

Term	Definition
CCSDS	Consultative Committee for Space Data Systems
ISO	International Standards Organization
TCP/IP	Transmission Control Protocol/Internet Protocol (basic communications standards of the Internet)

Framework Package 4: C++ Standard Library

Overview:

This package provides C++ language library functions as defined in the ISO standard (www.iso.org).

Description of basic functions:

Please refer to the ISO standard available at www.iso.org

Framework Package 5: Data Management and Transport

Overview:

Data Management Catalog (or Catalog) package provides the persistent storage service, and the database management services for State Variables and other data mediating components. The Catalog also serves as the interface to the Data Transport subsystem.

Access to persistent storage is provided through an ordinary file system interface (provided externally). The catalog acts as a configuration management wrapper layer around the underlying file system, and provides a data Product interface that allows applications to associate extended metadata with data objects, retrieve the metadata descriptions of data objects, convert between in-memory and serialized storage formats, and search for products based on metadata predicates.

Data Transport provides the abstract interfaces required of a MDS-compliant transport service. These interfaces define the data transport service's ability to enqueue and dispatch data products to and from a remote deployment. Although data transport mechanisms are not directly provided here, some interfaces and policy support classes are. The MDS data management and transport has been designed to move data products between deployments using the CCSDS File Delivery Protocol or similar transport protocol that can transport files with extended metadata. An adaptation's Transport Manager would be responsible for coordinating transport sessions, and queuing for transport those products that it finds in the catalog bearing metadata attributes that identify them as transportable. After the transport manager has sent a product, it can change the product's attributes in the catalog to reflect its changed status.

Description of basic functions:

Function Name	Description
Create collection	Create a new collection with the given name in the given collection.
Create container	Create a new container collection with the given name in the given collection if one doesn't already exist.
Find collection	Find only one matching collection by name.
Get sender	Accessor for the current product sender object.
Receive product	This is a callback method. When a product has been received and completely reconstructed, this method will be called to handle the product and decide what to do with it.
Report contents	Reporting method for listing the contents of the named collection.
Require collection	Same as create collection except that it will succeed even if the specified collection already exists.

Function Name	Description
Retrieve product	This function is a request to find a previously submitted product. The parameters to the function can include quite complex conditions on the metadata of the products to be retrieved. For example, a retrieve call might specify that only products created from 11:45 to 12:15 am, with size not more than 10kb, and appearing in certain collections are to be retrieved.
Send product	Request that the given data product be sent across an associated link using the transport service.
Send sendable	Issue a special query to find all the products that are marked for transport but haven't been sent yet and put them in the send queue.
Set sender	Configure the catalog to use the given product sender to support transport. The catalog accepts products for transport, but it itself does not interface to communications equipment, so it requires an object called a sender to which it can give products destined for transport.
Submit product	Adds a product to a catalog collection. In executing this function, the catalog makes a copy of the product, stores it to persistent storage, and stores meta data about the product in its internal data structures for later retrieval of the product. If a product has certain special meta data attributes, then this product is understood to be queued for Transport.

Glossary:

Term	Definition
Accessor	Function for retrieving an encapsulated variable.
Data catalog	Provides the persistent storage service, and the database management services for State Variables and other data mediating components. The catalog also serves as the interface to the Data Transport subsystem.
Data product (or Product)	The unit of data visible in the Catalog or in its interfaces. It's convenient to think of a product as a file with meta data attributes, and in workstation configurations of MDS, a product is stored as a file.
Data transport manager	Responsible for coordinating transport sessions with another deployment, and managing the transport of data products.

Term	Definition
Product sender	An object that provides an interface to communications software and which can send products out on the communications link view this interface.

Framework Package 6: Data Management Policies

Overview:

The Data Management Policies package, or Policy, provides a set of primitives for controlling data management and transport. For example, policies are can be used by value histories to implement rules like: 'Every time 5 new state values have been created, put the values in a product and submit the product for transport.'

Note: this is an MDS 2004 package that is not ported to MDS 2005.

Description of basic functions:

Function Name	Description
Policy actuator	A user-defined type that implements rule. Each policy actuator class has a condition and action. When a policy actuator object is evaluated, the object tests the condition—if it is true, the action is executed.
Condition	The part of a policy actuator object that is used to determine if the policy's action should be executed. Users of this package can define their own types of conditions, and still use the policy actuator class to implement the rule.
Action	The part of a policy actuator object that is executed when the conditions of the rule hold. Users of this package define their own action types. Thus, the policy package can be used to implement a large variety of rules.

Glossary:

Term	Definition
Policy actuator	User-defined type that implements a rule for triggering system activity.

Framework Package 7: Data Serialization

Overview:

The Data Serialization package provides functions needed to convert values to and from a series of bytes for portable and efficient storage and transmission over a communications link. This package also provides support for deserialization; i.e. the conversion of serialized data back to its initial value. These conversions are done in a platform-independent way; i.e. the value is preserved, even when platform-specific internal representations differ.

Description of basic functions:

Function Name	Description
Data input stream	Functions that convert serialized simple values, such as numbers and character strings, from a series of byte values into the values themselves.
Data output stream	Functions that convert simple values into series of bytes in a platform-independent way.

Glossary:

Term	Definition
Serialization	The conversion of a value from its internal representation in C++ or Java into a series of byte values, which can then be read by another piece of MDS software, regardless of which type of compute it is running on.
Deserialization	The opposite operation of serialization; the conversion of a series of byte values to their original C++ or Java values. For example, an 8-byte series may represent the real number 8.0. And the same 8-byte series would be translated, or deserialized, into the value 8.0 regardless of the kind of computer running the software that is reading the byte series.

Framework Package 8: Data Visualization

Overview:

The Visualization package provides rudimentary functions necessary to view the results of state queries. The capabilities of this package, include the ability to make graphical plots of state data, and to allow a user to configure a display based on values of the data. For example, a user may want the field in which a temperature value is displayed to turn red if the temperature goes above 180 degrees. This package also provides utility programs for converting binary query results into CSV (comma separated value) files, or for examining only a subset of the columns of data available in a binary result file.

Note: this is an MDS 2004 package that is not ported to MDS 2005.

Description of basic functions:

Function Name	Description
Plot results	This functionality appears as a 'Plot' button in several of the screens of the visualization applications. It causes a plot of the state values in the selected data as functions of time. Each different curve in the plot is given a different color, and there is a legend describing the color assignment to curve as well as unit information. The X-axis and Y-axis are labeled with unit information (e.g., a single plot might have a temperature in degrees Fahrenheit and a distance in meters. In this case, the Y-axis would have both units in its label).
Snapshot results	This function displays data not as a plot, but simply in a tabular-like format in which each data value is displayed with its own timestamp. These types of displays are normally used with real-time query data - then the user sees individual fields updated as fresh data is received. These kinds of displays can be color-coded to bring special attention to certain values for each field.
Export results	This function allows a user (or batch program) to write out binary query results into a CSV file, which can then be read by many applications, e.g. Microsoft Excel.
Read binary results	This function is used to allow applications to read in a previously stored file containing binary query results. After reading the data in, the user can plot, display, or export the data.

Function Name	Description
Subset results	This function, available either as a stand-alone application or as a window under the main Visualization application (called 'Query Manager'), allows the user to pick a subset of the columns of data available in a result for viewing or export. Sometimes queries have many columns of data, and a plot of them all is hardly decipherable, so it is very helpful to be able to select just a few columns for viewing.

Framework Package 9: Embedded Web Server and Client

Overview:

The Embedded Web Server, or EWS, provides a web server following the HTTP 1.0 protocol. This service receives HTTP requests via a TCP/IP port and dispatches them to other software components via a registration table. These software components register themselves at runtime with a text string corresponding to the resource string sent in the HTTP request. The software components then format an HTML response to send back to the requestor.

The Embedded Web Client (or EWC) provides a web client following the HTTP 1.0 protocol. EWC enables interaction with an embedded web server in the same way that browsers talk to web servers. This service sends HTTP requests via a TCP/IP port to a web server. The service then parses the HTML response for the user of the client.

Also included in this package is the MDS Directory Access Protocol, or MDAP provides a lookup service for storing and retrieving deployment addresses. MDAP servers run as a separate process in a multi-deployment scenario. At startup, each deployment registers its address with the MDAP server.

The package also provides an abstracted interface for a general purpose command interface to allow deployments to be externally commanded.

Description of basic functions:

Function Name	Description
Do dispatch	Dispatches a given HTTP request.
Http send	Sends an HTTP request to a particular address.
Http send to server	Sends an HTTP request to a server name looked up via Mds Directory Acces Protocol (MDAP).
Register CGI Function	Function called to register a software component for dispatching HTTP requests. The registration is with a string corresponding to the resource string passed in the server HTTP request.
Add entry	Adds a name/address pair to the MDAP database.
Retrieve entry	Retrieves an address from the MDAP database given and name.

Glossary:

Term	Definition
CGI	Common Gateway Interface. A standard used by web servers to communicate with other hosted processes.
EWC	Embedded Web Client.
EWS	Embedded Web Server.

Term	Definition
HTTP	Hyper Text Transfer Protocol.
HTML	Hyper Text Markup Language.
MDAP	MDS Directory Access Protocol.

Framework Package 10: Event Logging Facility

Overview:

The Event Logging Facility package, or ELF, provides a mechanism for recording messages generated when the system is operating. The ELF record, or log, may be used later by human operators to help diagnose problems or evaluate the performance.

An important feature of ELF is that the framework provides hooks to enable or suppress the generation of individual event messages. It does this by requiring that event messages can only be reported (published) by registered event Generators. Each generator identifies a source for a particular kind of event with a particular severity, and provides the filtering control apparatus for suppressing the reporting of that event.

ELF includes three sub packages:

- An interface package for reporting events
- A package that specializes the reporting package for initialization, finalization, or other special cases.
- A package for storing and transporting recorded messages. In particular, it provides control for storing and filtering messages. It also provides a mechanism for specifying which messages are generated.

Description of basic functions:

Function Name	Description
Generate message	This is the generic function that can be called to generate an event message. The argument contains the message. The method will add a time tag to the event message and will decide whether or not to save it based on internal filtering options.
Set filters	Set filtering options for a particular kind of event message. Messages can be filtered to suppress on a duration basis (permit messages to be generated no more often than the given duration), interval basis (suppress all but one of every N instances), or a severity basis (suppress messages whose severity is less than X). A given message type can also be disabled entirely.
Initialize	Initialization options primarily include specification of a class (method) for handling messages. Elf can be configured to send messages to a file, to a console, or to a handler event that will convert the event messages into data products for transport to another MDS deployment.

Glossary:

Term	Definition

Term	Definition
ELF	Error Logging facility

Framework Package 11: Goal Elaboration Language

Overview:

The Goal Elaboration Language package, or GEL, provides a textual language for specifying goal networks and elaborations. We also refer to the processor for this language as "GEL".

Note: this is an MDS 2004 package that is not ported to MDS 2005.

Description of basic functions:

Function Name	Description
Define goal	Define a goal in GEL. A goal definition specifies such things as the underlying state variable, state constraint, and the time points of the goal, as well as any tactics for maintaining the goal. These tactics may include other sub-goals and temporal constraints. For example, other values may be passed as parameters to be used inside of tactics.
Make goal	Create a new goal from a GEL goal definition. Any number of goals can be created from a single goal definition.
Elaborate goal	Once a goal is created, it may be elaborated as a separate step. Elaboration entails trying the tactics for maintaining the goal, until one is found. Elaboration relies on package GNET to do much of the work, Gel acting as an interface.
Define subnet	A subnet is a network of goals and temporal constraints, intended for insertion into the overall goal network. It is similar to a tactic, the difference being that it does not reside within some goal definition.
Insert subnet	Insert a defined subnet into the network.
Make Xgoal	Make an executable goal. Executable goals ("Xgoals") are imported from lower level code. When an Xgoal is made, it is immediately inserted into the network, in contrast to a goal, which does not affect the network until it is elaborated.
Make time point	Create a new time point, for example one to be used in conjunction with one or more Xgoals.
Make temporal constraint	Create a new temporal constraint, for example one to be used in conjunction with Xgoals to constrain their execution start or finish.

Function Name	Description
Define value	In addition to the items mentioned above, many other kinds of values can be defined in GEL, which provides the computational capability of a general purpose programming language.
Compute value	GEL provides an expression language that can be used to compute arbitrary arithmetic and symbolic values. The syntax for this aspect is similar to conventional languages such as Lisp or Scheme.
Define function	GEL provides for defining arbitrary computational functions, either in GEL itself or by binding to a function defined in the implementation language, currently C++. These functions are bound to names so as to be useable within GEL.
Load file	GEL can specify that files containing additional GEL expressions are to be loaded and interpreted.
Trace	Turn on or off a trace of GEL execution.

Glossary:

Term	Definition
Binding	The association between a name and a value or function.
GEL	Goal Elaboration Language.
Goal	A constraint on a state variable that is to be maintained between two time points.
Time point	A symbolic point in time. The actual time represented by a time point is determined during execution, and is constrained by temporal constraints with other time points.
Temporal constraint	A constraint between two time points that specifies a minimum and maximum delay time between those time points.
Xgoal	Executable goal.

Framework Package 12: Graph

Overview:

The Graph package provides data structures and algorithms needed to represent and work with discrete graphs. A discrete graph is a mathematical structure defined in terms of a finite set of vertices and a finite set of edges. A graph edge defines a binary relationship between two vertices. Such graphs are pervasive throughout computer science—working with graphs is a fundamental aspect of this field.[1]

The MDS Graph framework is a straightforward implementation of the concepts and techniques described in the popular computer science textbook: "Introduction to Algorithms" by Cormen, Leiserson and Rivest. There are two differences from the treatment of graphs in this text: 1) caching and 2) membership graphs. The textbook focuses on algorithms but does not address the computational issues involved in repeated uses of such algorithms. A simple optimization consists in storing the results of the first execution of an algorithm and reusing these results for subsequent requests for the same algorithm. A membership graph is a specialized application of a discrete graph that tracks the membership of each vertex to different groups of vertices.

There are three kinds of graphs in this package: directed, undirected, and equivalence. Each kind is one of two varieties: a topological graph or a data graph. A topological graph represents only the topology of vertex/edge connectivity. A data graph is a topological graph with the addition of user-defined data for each vertex and edge.

Description of basic functions:

Function Name	Description
Graph editing	Basic editing functions include adding & deleting vertices & edges. To optimize query performance, graphs maintain a query cache. To edit a graph, it must first be "opened" for editing (i.e., it can be edited but not queried). A graph cannot be queried until it is "closed" to editing, at which time the cache is reset.
Graph query	Access properties of the graph (number of edges, vertices). Browse the graph topology (vertices, edges, paths through the graph).
Graph algorithms	Topological sort, Depth-first search (nearly a verbatim adaptation from Cormen et al).

[1] Graph theory is part of the standard computer science curriculum in either advanced undergraduate coursework or first year graduate studies in computer science.

Glossary:

Term	Definition
Graph	A data structure that has a collection of vertices and edges. In a topological graph, vertices and edges are identified with a unique number. A data graph is like a topological graph but it associates a user-defined data item to each vertex and to each edge.
Vertex	An item that is part of the graph.
Edge	An item that represents a relationship between two vertices in a graph.

Framework Package 13: Graph State Variable

Overview:

The Graph State Variable package, or GSV, provides a mechanism for defining relative states as pair-wise directed relationships between nodes in a graph. GSVs are a general graph based state representation that (1) can derive a state's value by combining relationships, (2) produces different results for different derivation paths, and (3) handles changes to topology and relationships between nodes.

Note: this is an MDS 2004 package that is not ported to MDS 2005.

Description of basic functions:

Function Name	Description
Close	Completes the modification of a graph state variable's topology.
GetRelationship	Retrieves a value for the relationship between two nodes in a graph state variable.
Open	Prepares a graph state variable's topology for modification.
RecoverRelationship	Restores a value from persistent store for the direct relationship between two nodes in a graph state variable.
UpdateChildren	Adds constituent graph state variables to a composite graph state variable as specified in a read/write parameter file.
UpdateRelationship	Stores a value for the direct relationship between two nodes in a basis graph state variable.

Glossary:

Term	Definition
Frame	A frame is point of reference.
GSV	Graph State Variable.
Node	A node in a graph state variable is an abstraction of a frame.
Direct relationship	A direct relationship within a graph state variable is abstracted as an edge between two nodes, where the edge references the value of the relationship between the two frames represented by the two nodes.
Derived relationship	A derived relationship within a graph state variable is a relationship computed by concatenating the relationships along a path that has 3 or more nodes.
Basis graph state	A basis graph state variable is comprised of direct

Term	Definition
variable	relationships that are estimated locally.
Proxy graph state variable	A proxy state variable is comprised of direct relationships that are copied from a remote location.
Composite graph state variable	A composite graph state variable is comprised of graph state variables that are connected by sharing one or more nodes.
Constituent graph state variable	A constituent graph state variable is a graph state variable that is an element of a composite graph state variable.

Framework Package 14: Hardware Adapter

Overview:

The Hardware Adapter package provides base classes for command and measurements. These classes can be further specialized to suit given hardware devices.

A “hardware adapter” is an executable software component that provides uniform interfaces to a hardware device or its simulation. Interfaces are defined for submission of commands to actuators, querying of measurements from sensors, and policy-driven management of command and measurement histories. A hardware adapter may extend the functionality of a raw hardware device to provide a more convenient basis for monitoring and control, but it must not hide information needed for fault protection.

There is no hardware adapter base class in the framework because there is no common behavior. A hardware adapter adaptation must implement an interface to accept a command of the subtype(s) it can accept. Similarly, it must implement an interface to provide a measurement of the subtype(s) it generates and a command of the subtype(s) it can accept. The hardware adapter adaptation must implement one or more value histories that store its measurements and commands.

Note that this package is implemented as a sub-package within the state package.

Description of basic functions:

Function Name	Description
Submit command	This method is invoked by a controller component to submit a command. The method should do little more than store the argument command in a value history, so that the command can subsequently be enacted by the run method. The command may supersede, amend, or complement previously received commands.
Get command	This method is invoked by an estimator component. It returns a previously submitted command stored in a hardware adapter's value history.
Get measurement	This method is invoked by an estimator component. It returns a measurement stored in a hardware adapter's value history.

Glossary:

Term	Definition
Command	A time-tagged outgoing directive to change one or more physical states in the system under control. Issued by controllers to hardware adapters. May be used by estimators as evidence for estimating state variables.

Term	Definition
Measurement	Provides time-tagged evidence about one or more physical states in the system under control for a moment in time. May be a science observation. Provided by a hardware adapter. Used by estimators as evidence for estimating state variables.
Hardware adapter	Executable software component that provides uniform interfaces to a hardware device or its simulation. Provides a measurement and command interface between the hardware of the system under control and the control system. Keeps one or command and measurement value histories.

Framework Package 15: Image

Overview:

The Image package provides base classes for creating, reading, writing, and accessing elements of image data structures based on the measurement framework. This allows image data to be manipulated as measurements produced by a camera hardware adapter, stored in a value history, and stored and transported as data using the serialization and data management framework services.

This package supports the JPLPic image format.

Description of basic functions:

Function Name	Description
Read image	Creates an image by reading a file containing image data.
Write image	Writes image data to a file

Framework Package 16: Initialization and Finalization

Overview:

The Initialization and Finalization, or INIT, package provides the mechanism for managing dependencies among singleton classes. A singleton class is a special class type that can have one and only one class instance at runtime.

The singleton is one of the many design patterns documented in "Design Patterns" by Erich, Gamma, Johnson and Vlissides. The Init framework is a substantial improvement over the basic singleton pattern for rigorously handling initialization, finalization issues of singletons in a way that provides both metrics on pattern usage, measures of testing complexity and a capability for exhaustive testing. The Init framework is highly portable across C++ compilers and platforms because it delays the initialization of all application-level singletons until the C++ runtime system has been fully initialized (including exception handling facilities) and forces the finalization of all application-level singletons to occur before the C++ runtime system looses its exception handling facilities.

Description of basic functions:

Function Name	Description
Singleton pattern	There are several variations of the basic pattern available for regular C++ classes as well as for parametric classes also known as C++ template classes.
Instance	The framework includes an instance method for each user-defined application of one of the singleton patterns. This instance method performs runtime checks relative to the proper initialization and ordering of the singleton.
Topological sort	The Init framework sorts all singletons according to the topological sorting of their partial dependencies. This is a straight application of the Depth-First Search algorithm from "Introduction to Algorithms" by Cormen, Leiserson, Rivert. This algorithm is further instrumented to allow the framework to generate all possible topological sortings of the dependencies. This generative capability is essential for rigorous verification and validation of software.
Pseudo-static initializer	A pseudo-static initializer is an object that uses the "Resource Allocation Is Initialization" principle to ensure that all singletons will be initialized when the object is constructed and that all singletons that have been initialized will be finalized before the object is destroyed.

Glossary:

Term	Definition
Singleton	A C++ class type that has a static method (usually called "instance") that retrieves the unique instance of this class type. See "Design Patterns" by Erich, Gamma, Johnson and Vlissides for a complete description.
INIT	Initialization package
Topological sort	An algorithm from graph theory intended to find the total order of a set of items that have only partial ordering constraints among them.
Initialization & finalization	All software that runs on hardware is subject to an initialization phase before the software actually runs and a finalization phase after the software has completed its execution. The C++ standard provides weak guarantees about this phase of a program execution. These guarantees are insufficient to meet the needs of robust software. INIT includes a solution that provides strong guarantees about this process that either address critical software requirements or greatly facilitate their resolution.

Framework Package 17: Math

Overview:

The Math package provides common mathematical computations for MDS frameworks, adaptations, and deployments. It includes object definitions and methods for standard library limits class, time intervals, 3-vectors, quaternions, Euler angles, normal distributions, polynomial functions, Taylor series, linear algebra algorithms, and math exceptions. The MONTE project provided many of the functions required to implement 3-vectors, quaternions, Euler angles, and linear algebra algorithms.

Description of basic functions:

Function Name	Description
Contains	Returns true if the interval contains a member value.
Is superset of	Returns true if this interval contains another interval.
Is contained by	Returns true if this interval is a subinterval of another interval.
Is equivalent to	Returns true if this interval is equivalent to another interval.
Has null intersection	Returns true if this interval has no overlap with another interval.
Unit	Computes a unit vector (and time derivatives).
Transpose	Transposes a matrix.
Subtract	Computes the component-by-component difference of a matrix or a vector.
Project	Computes the projection of a vector onto another vector.
Orthogonal	Computes the component of a vector that is orthogonal to another vector.
Multiply	Multiply all elements of a vector (or a matrix) by a scalar.
Max magnitude element	Computes the maximum magnitude (absolute value) element of a vector.
Matrix vector multiply	Multiply a matrix times a vector.
Matrix transposed vector multiply	Multiply a matrix transposed times a vector.
Matrix matrix multiply	Multiply a matrix times a matrix.
Matrix transposed matrix multiply	Multiply a matrix transposed times a matrix.
Invert 3	Computes the inverse of a 3x3 matrix.

Function Name	Description
Magnitude	Computes the vector magnitude (and time derivatives).
Identity	Sets a square matrix equal to the identity matrix.
Dot	Computes the vector dot product (and time derivatives).
Determinant 3	Computes the determinant of a 3x3 matrix.
Cross	Computes a vector cross product (and time derivatives).
Combine	Computes a linear combination of two vectors.
Angle	Computes the angle (in radians) between two vectors.
Add	Computes the component-by-component sum of 2 vectors or 2 matrices.
Taylor Series	Constructs a Taylor series function from a list of coefficients.
Compute value of Taylor series	Evaluates a Taylor series function at a point in time.
Compute n^{th} derivative of Taylor series	Evaluates a derivative of a Taylor series function at a point in time.
Polynomial function	Constructs a polynomial function from a list of coefficients.
Compute value of polynomial function	Evaluates a polynomial function at a point in time.
Get n^{th} derivative of polynomial function	Evaluates a derivative of a polynomial function at a point in time.
Normal distribution	Constructor for creating a normal distribution of a given mean value and standard deviation.
Get probability of normal distribution	Returns probability that a value is within a given range of a normal distribution.
Get probability of standard normal distribution	Return probability that a value is within the range [low high] in a standard normal distribution. This class supports fast probability calculations in a standard normal distribution via table look-up rather than via numeric integration.
Get nz	Return probability that a value lies in the range from -infinity to v in a standard normal distribution.
Derivative	Constructs a quaternion (and its derivative) corresponding to a coordinate transformation obtained by rotating about a coordinate axis at a specified angular rate; constructs a quaternion (and its first and second derivatives) corresponding to a coordinate transformation obtained by

Function Name	Description
	rotating about a coordinate axis at a specified angular rate and acceleration.
Quaternion	Constructs a quaternion from real and imaginary parts; from a series of three axes and Euler angles; from a 3-1-3 series of Euler angles (called an "MDS Constructor", developed for EDL GNC applications); from a rotation about a coordinate axis that is one of the purely imaginary basis quaternions, i, j, or k.
Euler	Constructs a 3-1-3 Euler angle rotation from 3 angles.

Framework Package 18: MDS Standard Library

Overview:

The MDS Standard Library, or MdsStd, package provides utility functions that are used frequently throughout the MDS frameworks. MdsStd includes low-level utility classes and interfaces that are not already provided by our operating-system wrapper, the language's standard library, or the programming language itself. For example, smart pointers, file manipulators, and case-insensitive string comparison. Much of the functionality of this package is very similar to that found in several well-known C++ extension packages, such as Loki or Boost.

Included in the MDS Standard library are exceptions, a standard mechanism for defining exception types and creating exception variables. This package simply defines a base class for all MDS exception (error) messages. Exception can be used directly, or extended in other MDS classes to define all exceptions that can be generated at runtime from MDS code.

Having a common base class allows the exceptions generated at runtime to be more easily identified as having come from MDS code (as opposed to some third-party software). The MDS exception base class is derived from the standard library exception class and provides no additional functionality.

Description of basic functions:

Function Name	Description
Caseless string	This function makes it easy to compare two character sequences without regard to the case of the letters in the sequences.
Reference-counting pointer	This function makes it easy for several different pieces of software to have access to a value without having to have their own copy of the value, and to automate releasing the memory associated with the value when no more references to it exist.
Holder-pointer	This function automates the releasing of dynamically allocated memory occupied by a variable when that variable is no longer needed.
Context string	This function makes it easy to read and manipulate character strings, such as a path string, that imply a hierarchy. Sub functions include breaking such a string into its component pieces.
Stream tokenizer	This function makes it easy to accept a stream of characters and have it automatically broken into single words.
Print formatter	This function defines a simple interface for formatting text reports.

Function Name	Description
Usage timer	This functionality makes it possible to time the execution of a function or code block.
Exception	Base class for all MDS exceptions. Having a single Base Class of all thrown objects in MDS simplifies exception handling.

Glossary:

Term	Definition
MdsStd	MDS Standard Library package
Path String	A string that defines a folder in a computer's disk. For example, the string `'C:\windows\system\Program Files'` is a path string, and it implies a hierarchy: Folder 'C' contains folder 'windows', which in turn contains folder 'system', and so on.
Pointer	In C++, a variable that contains the memory location, or address, of another variable.
Class	In C++ or Java, a user-defined type that incorporates functions and data variables. A class may be extended to define refinements of the original class, in which case the extensions are said to be 'derived' from the original ‘base’ class.
Base class	A class from which other classes are derived. Base class are usually intended to establish the basic functionality or role of an entire hierarchy of classes.

Framework Package 19: MPE Coordination

Overview:

Provides a goal execution engine and coordinates elaboration and goal checking as part of the Mission Planning and Execution (MPE) function. It contains the Xgoal checker, the executive, and the elaboration manager.

The Xgoal checker is a runnable component which is responsible for monitoring the status of Goals whose associated Xgoals are currently active (executing) in the active Xgoal Network. Goals become active when any of their associated Xgoals begin execution (when their opening Time point is fired). At that point, the Xgoal checker begins periodically evaluating the status of each associated goal by way its Xgoal.

The executive is a runnable component responsible for managing the execution of the Xgoal network. Specifically, this includes the task of dispatching new Xgoals to achievers by way of their state variables as time points are "fired", and "firing" time points as time advances and state variables change. It also promotes scheduled Xgoal networks to be executed.

The elaboration manager is a runnable component that runs elaborators. It checks the data catalog for new goal net products from the ground and gives the executive a scheduled Xgoal network to be promoted for execution.

Note that this package is implemented as a sub-package within the state package.

Description of basic functions:

Function Name	Description
Elaboration	Expands a goal into a network of supporting sub-goals needed to achieve the goal.
Scheduling	Determines the execution order of goals on state variable timelines so that all temporal and state constraints can be met.
Promotion	Installs a scheduled goal net to be executed.
Time point firing	Fires time points when temporal and state constraints (Xgoals) are met.
Xgoal dispatching	Sends an Xgoal to a state variable for execution when the Xgoal's starting time points fire.
Xgoal checking	Verifies that executing Xgoals are still satisfiable and triggers Xgoal failure when they are not.
Execute Xgoal network	A executive interface to execute the currently promoted Xgoal network.

Glossary:

Term	Definition
Fired time point	A time point in an executing goal network whose pre- and

Term	Definition
	post-condition have been satisfied so that all incoming goals can be completed and all outgoing goals can begin executing.
Promotion	Installing a scheduled goal net to be immediately executed.

Framework Package 20: Naming Services

Overview:

The Naming Services package is intended to provide mechanism by which components and other runtime objects may be found and addressed within a deployment. This package provides an easily accessible capability for storing values with an associated name in a global registry. Once values are stores in a registry, they can be used by naming service functions.

This package also provides the ability to create 'contexts' or local regions. Each context contains a local registry for the value-names pairs can be that allows the registration of values under a name, but these associations are defined only within the context.

Description of basic functions:

Function Name	Description
Register object	Creates an association in the registry between the given object and a name. The name can be supplied by the caller, or can be generated by this package. In either case, the name must be unique. The object can be subsequently looked up in the registry by name. The registration can be specified to be in the global registry, or in a registry that is contained within a context.
Deregister object	Removes the association between the given object and its name in the registry. The object must have previously been registered using the 'register object' function.
Get object	Given a name, finds and returns the object that is registered under that name.
Create context	Creates (defines) a new context, optionally as a sub-context to an already-existing context.

Glossary:

Term	Definition
Context	A scope or limit of definition for object registries. An example is a folder in a computer disk: the folder provides a context in which the names of the files in the folder are defined. Filenames must be unique within a single folder, but different folders may contain files of the same name.
NOR	Naming Service package
Object	In object-oriented programming, an instance of a specific, usually user-defined, class.
Object Registry	A set of associations of a name to an object.

Framework Package 21: Performance

Overview:

The Performance package provides tools to gather and analyze performance metrics. These tools include heap size monitors, timers, and a tool for acquiring CPU instruction-level measurements.

The Papi monitor provides controls to access hardware events, such as cache hits/misses etc., for performance monitoring. It uses the Performance Application Programming Interface (PAPI) externally-provided library. PAPI provides unified interfaces to most major microprocessor hardware. This is an optional package that is not compiled into a build by default. See http://icl.cs.utk.edu/papi for more details on PAPI.

Description of basic functions:

Function Name	Description
start timer	Start the timer
stop timer	Stop the timer
generate timing report	Produce a file that contains timing data and statistics for each timer.
start heap monitor	Sets a heap level
stop heap monitor	Compute heap usage since the start heap function
generate heap usage report	Produce a file that contains heap usage information and statistics for each heap monitor.

Framework Package 22: Physics

Overview:

The Physics package provides a set of common physics objects and computations. It contains object definitions for Cartesian, cylindrical, and geodetic coordinate systems and methods for conversions between them. In addition, it contains representations and methods for positions and rotations, and their first and second derivatives, and methods for rotating positions and their derivatives. The Physics package contains object definitions and methods for ellipsoids and physics exceptions. Finally, it contains the six degree-of-freedom (6DOF) transformation class, and methods to apply, invert, combine, and propagate 6DOF transformations.

Description of basic functions:

Function Name	Description
6DOF	Constructs a six degree-of–freedom transformation from either a rotation and acceleration objects; a quaternion, angular velocity, angular acceleration, cartesian position, cartesian velocity, and a cartesian acceleration; or a quaternion, angular rate, angular acceleration, spherical coordinates, spherical rate, and spherical acceleration.
6DOF Invert	Constructs a six degree-of–freedom transformation that is the inverse transformation.
6DOF Concat	Constructs a six degree-of–freedom transformation that is equivalent to successive applications of two different transformations.
6DOF Propagate	Constructs a six degree-of freedom transformation from another by propagating the position and rotation in time assuming a zero angular acceleration.
Cylindrical system to Cartesian	Converts a cylindrical position, velocity, and acceleration into Cartesian coordinates.
Cylindrical system from Cartesian	Converts a Cartesian position, velocity, and acceleration into cylindrical coordinates.
Geodetic system to Cartesian	Converts a geodetic position, velocity, and acceleration into Cartesian coordinates.
Geodetic system from Cartesian	Converts a Cartesian position, velocity, and acceleration into geodetic coordinates.
Spherical system to Cartesian	Converts a spherical position, velocity, and acceleration into Cartesian coordinates.
Spherical system from Cartesian	Converts a Cartesian position, velocity, and acceleration into spherical coordinates.
Rotation	Constructs a rotation from a coordinate axis, angle, rate, and acceleration; an axis (eigenvector of the rotation),

Function Name	Description
	angle of the coordinate rotation about that axis, rate, and acceleration; a quaternion and its derivatives; a quaternion, angular velocity and angular acceleration; a 3x3 rotation matrix and its first and second derivatives; a sequence of Euler angles and their derivatives.
Is within	Determines if one rotation is close to another.
Angle and axis and derivatives	Computes the angle and axis (Eigen vector) of a rotation and the axes and values of its derivatives.
Euler angles and derivatives	Computes the angles and their first and second derivatives for an Euler factorization corresponding to a user specified sequence of coordinate axes.
Quaternion and derivatives	Computes the quaternion associated with a rotation as well as the quaternion's first and second derivatives
Propagate	Computes the rotation propagated from a given rotation assuming constant angular velocity.
Matrices	Computes the rotation matrix and its first two derivatives corresponding to a given rotation.
* - Rotation concatenation and application operator	Computes the rotation corresponding to two successive rotations; or computes a rotated position, velocity, and acceleration.
Invert	Computes the multiplicative inverse of a rotation
Apply inverse	Applies the inversion of a rotation to a position, velocity, and acceleration.
Near point	Determines the point on the surface of an ellipsoid that is closest to a specified point.
Derivatives to normal	Determines the unit normal and its first and second time derivatives at a point with known velocity and acceleration on the surface of an ellipsoid.
Base of normal	Given a non-zero input vector and its first two time derivatives, determines the points on the surface of the ellipsoid whose outward pointing normals are parallel to the input vector and its derivatives.

Glossary:

Term	Definition
6DOF	Six degrees of freedom

Framework Package 23: Sequential Estimation

Overview:

The Sequential Estimation package provides an application programming interface (API) that facilitates developing extended Kalman filters (an algorithm reported extensively in the open literature since the early 1960's) for real-time and post-processing applications. The APIs provide a generic mechanism for estimating parameters, based on streams of measurement data, measurement sets, and dynamics models.

In practice, this package is co-compiled with other user-supplied software modules that mathematically compute the measurements and system dynamics of particular interest to the user. The resulting product of the co-compilation is a new extended Kalman filter routine, program, or application.

Note: this is a package that is not ported to MDS 2005.

Description of basic functions:

Function Name	Description
Add matrices	This function adds two matrices together and returns the sum.
Multiply matrices	This function multiplies two matrices together, and returns the product.
Convert quaternion to Euler angles	Converts from a quaternion attitude representation to a Euler angle attitude representation.
Convert Euler angles to quaternion	Converts from a Euler angle attitude representation to a quaternion attitude representation.
Propagate quaternion in time	Given the attitude quaternion of a body at a start time, computes the attitude quaternion of the same body at a different time, by multiplying the time rate of change of the body's attitude quaternion, by the change in time, via standard integration steps.
Propagate covariance matrix	Given the covariance matrix at a start time, computes the covariance matrix at a different time, using appropriate user-provided matrices containing the user's system dynamics models, and uncertainty/assumed error in the dynamics models.
Propagate state vector	Given the state vector at a start time, computes the state vector at a different time, using appropriate user-provided matrices containing the user's system dynamics, in the form of the first derivative in time of the state vector.

Function Name	Description
Kalman-update covariance matrix	Given the covariance matrix at some time that a filter measurement is made, representing the uncertainty in the state vector at that time but without the information in the filter measurement, this function computes the covariance matrix (by standard Kalman update referenced in the literature) representing the improved uncertainty after applying the information in the filter measurement.
Kalman-update state vector	Given the best estimated state vector at some time that a filter measurement is made, this function applies the standard Kalman update (referenced in the literature) to make a linear improvement "step" to the estimated state vector, based on the information in the filter measurement. Value of the improvement "step" depends on the user-supplied formulation for the mathematical model of the filter measurement.

Glossary:

Term	Definition
Covariance matrix	Associated uncertainty of each parameter in the state vector, and correlated uncertainty, in the form of an *n*-by-*n* matrix, where *n* is the dimensionality of the state vector. An always-present by-product of any extended Kalman filter.
Dynamics model	The mathematical expression of the time evolution of a system. For example, the mathematical expression for the sum of the shear and normal forces acting on the wheels of a vehicle, along with the weight of the vehicle, and the wind force pushing on a vehicle, which gives the rate of change of the velocity of a vehicle. A user-defined input to any extended Kalman filter; therefore, not supplied by the sequential estimation framework.
Euler angles	Another standard representation of the attitude ("pointing direction") of a body in space, with the property of having well-known, intuitive physical meaning, in terms of roll, pitch, yaw about a set of axes. Euler angles have problematic mathematical singularities (regions where numerical values go to infinity, even though expected physical value has not gone to infinity) when propagated in time.
Matrix	Standard representation of an ordered set of numerical values, in an arrangement of n rows by m columns (n and m integer, greater than zero). Standard rules of linear

Term	Definition
	algebra apply to adding, subtracting, multiplying and dividing matrices.
Filter Measurement	Numerical value of the output of a sensor, said output being mathematically represented as a function of the state vector. This is not the same as the MDS measurement, but would typically include the information from one or more MDS measurements with the same time tags. A user-defined input to any extended Kalman filter; therefore, not supplied by the sequential estimation framework.
Quaternion	Standard representation of the attitude ("pointing direction") of a body in space, with mathematical property of not having mathematical singularities when propagated in time.
State vector	Set of n parameters being estimated with the extended Kalman filter – defined by user. Main product of any extended Kalman filter. Not a state variable, but may be used to estimate one or more state variables.

Framework Package 24: Simulation

Overview:

The Simulation package provides tools that simplify the job of building simulation components.

Note: this is an MDS 2004 package that is not ported to MDS 2005.

Description of basic functions:

Function Name	Description
Get data	This function allows deployments to get the sensor data from a simulated sensor device.
Command	This function allows deployments to issue commands to the simulated actuator device.
Run sim device	This function allows the thread scheduler to execute a sim device component for a cycle.
Set request	This function allows the sim device via sim access bridge to communicate with third-party software. It sends a request to the third-party software.
Get response	This function gets a response back from third-party software to a sim device.
Translate request	This function translates a request made by a set request function call into the form required by third-party software tool to communicate.
Translate response	This function translates the response returned by third-party software into an internal response data structure needed by the get response command.

Glossary:

Term	Definition
Sim	Truncated name for simulation.
Sim access bridge	A Sim access bridge contains a discrete value history for transactions, which record requests from and responses to a hardware adapter in a different deployment. In addition to providing this value history, a sim-access-bridge component provides the communications channel with a deployment, interpreting as needed commands and queries to a sim-device component.

Term	Definition
Sim devices	Sim devices are state generators, with the additional capability to provide an interface for commanding and data retrieval. Sim devices may either contain internal device modeling or communicate with third-party dynamic simulation tools in order to simulate discrete device states.
State generator	Analogous to remote estimators, they generate state functions with which to update state variables. A state generator may contain either internal dynamics modeling or communicate with third-party dynamic simulation tools in order to simulate continuous physical states.
Sim model bridge	A sim model bridge is a component by which sim communicates with external simulation models or third party tools. This communication is performed using a remote-procedure-call style interface.

Framework Package 25: State

Overview:

The State package provides a mechanism for representing state variables and their values. The concept of "state knowledge" is fundamental to the MDS architecture and refers to "what we know and how well we know it."

State knowledge encompasses information such as device operating modes, device health, resource levels, attitude and trajectory, temperatures, pressures, etc, as well as environmental states such as the motions of celestial bodies and solar flux. Such information is maintained in state variables that serve, in a sense, as "Grand Central Station" because of the many activities that use state information, including estimation, control, planning, and telemetry.

This package provides general-purpose classes and methods for representing, accessing, and managing state knowledge. The three basic elements of state knowledge are state variables, state functions, and state values. State variables provide uniform methods for querying and updating state knowledge as well as managing the data stored within. State functions, which describe how a state's estimated value varies with time, are used to update state variable timelines. State values are returned by state variables in response to a query for a particular instant of time. The classes for state functions and state values are abstract base classes that users extend to represent data in a suitable form; thus, this package does not try to prescribe user data formats.

All state variables support the following interfaces: 'goal execution', and 'goal scheduling'.

This package provides achievers, goal-driven executable components that strive to achieve executable goals. State controllers and state estimators are types of goal achievers; the former strive to achieve a constraint on the value of a state variable and the latter strive to achieve a goal on the quality of knowledge in a state variable.

All achievers support the following interfaces: 'runtime execution', 'goal execution', and 'goal scheduling'. In addition, controllers support the 'command submit' interface, and estimators support the 'state update' interface.

This package also provides the framework used to build and execute temporal constraint and goal networks, as well as elaboration and scheduling capabilities for these networks.

Description of basic functions:

Function Name	Description
Get state	A state variable interface whereby a state variable returns a state value for a specified instant in time. A legitimate value is "unknown", meaning that the state variable currently contains no information for that instant in time.
State update	A state variable interface whereby an estimator updates the state value of a state variable for the time interval specified in a supplied state function.

Function Name	Description
Notify listener	This interface notifies listeners that the state variable has been updated.
Elaborate goal	Create supporting goals as specified by a dependent goal.
Schedule goal network	Produce Xgoal network for execution. Temporally constrain conflicting goals so they do not overlap in time, and merge consistent goals into Xgoals that do not conflict. Order time points of affecting states with respect to the time points of affected states, and compute projections. Check Xgoals for achievability (i.e. that the projections for all Xgoals are consistent with their corresponding merged goals).
Goal scheduling interfaces	Each state variable provides interfaces used during scheduling of executable goals to: determine if a goal-to-goal transition is achievable, project the effect of a goal on the state, and determine if a goal is achievable. State variables consult any achievers they have to perform these operations.
Propagate network	Calculate the temporal consistency of the temporal constraint and goal network during network scheduling. Calculate what time points can fire in a network during goal network execution based on temporal constraints and the current time.
Goal execution interfaces	Each state variable provides interfaces used during goal net execution: check if an Xgoal is ready to transition given current state, start execution of the Xgoal, and if a goals is still satisfiable. A state variable having achievers will consult with them to determine if an Xgoal is ready to transition, and will forward them an Xgoal to be executed on their start Xgoal execution interface.
Run achiever	Interface for the component scheduler to run an achiever during goal execution.

Glossary:

Term	Definition
Goal	A desired state over a particular temporal interval.
Temporal constraint	The specification of a flexible temporal relationship between time points in a network.
Temporal constraint network	A directed graph whose edges are temporal constraints and nodes are time points.

Term	Definition
Goal network	A directed graph whose edges are goals and temporal constraints and whose nodes are time points.
Executable goal (Xgoal)	The result of merging goals during goal network scheduling. Contains a projection.
Xgoal network	A scheduled goal network that contains an Xgoal time line for each state variable. An Xgoal network is executable.
Projection	A state prediction corresponding to an Xgoal.
State value	The estimate for of a state variable at an instant in time. The uncertainty in the estimate must be represented in some form as part of the state value. Function ‘get state’ returns a state value.
State function	A function of time, bounded by a start time and end time that describes how a state varies over time, if at all. Function ‘update state’ takes a state function as an argument.
State variable	Provides uniform methods for querying and updating state knowledge as well as managing the data stored within.
Achiever	A runnable estimator or a controller component that takes action to change or maintain state as specified in Xgoals.
Estimator	An achiever for a state variable that executes knowledge goals to determine state.
Controller	An achiever for a state variable that executes control goals to change or maintain state.

Framework Package 26: State Query

Overview:

The State Query package provides the functionality to query state and measurement histories. This functionality includes the ability to submit queries from either text files or from a GUI application, and to receive query results in batch or real-time modes. The package also contains helper classes to make it easy to extend the query functionality for new kinds of state-data types, and an application to help a user generate an adaptation of the query library for a new kind of state data.

The package contains C++ and Java elements, and includes the functionality to communicate between C++ and Java deployments.

Note: this is an MDS 2004 package that is not ported to MDS 2005.

Description of basic functions:

Function Name	Description
Create query	Implemented by a Java GUI application window, this function allows a human user to interactively build a query, by selecting a time range, type of query, and a set of states to be queried. This application can also store or read queries in text form to/from a file.
Submit query	This functionality allows the submission of a query for processing. Submission can be done by a human user via the query builder application, or by sending an Http command to a batch-mode Java application, which in turn locates the C++ deployments involved in the query and sends each of them the query.
Execute query	In C++ deployments, this function accepts a query from a Java application via the Embedded Web Server (in the form of an Http message). This function reads and analyzes the query, searches for the state objects to be queried, and retrieves the data. It then builds a response message containing the query results and sends it back to the requestor.
Generate queries	This functionality automatically searches in a C++ deployment for queryable state and measurement histories. It generates and executes a query for every one found. This is especially useful in testing.
Search states	This function is implemented as a Web command in a C++ deployment. It searches the deployment to find all queryable states, and returns the list of these to the requestor. This function is used by the Java query builder application to discover the list of available states from

Function Name	Description
	which the user can select.

Framework Package 27: Task Scheduler

Overview:

The Task Scheduler framework package provides interfaces for multi-threaded software applications. The fundamental aspect of this package is to define software interfaces to support coordinated periodic execution.

Description of basic functions:

Function Name	Description
Periodic interface	Interface for a software element that will be called periodically
Background task interface	Interface for a software element that is executed on unused CPU resources. Includes a yield mechanism for returning control back to the task scheduler.
Creating and Scheduling Rate Groups	Mechanism for grouping periodic tasks into commonly scheduled groups

Glossary:

Term	Definition
Deployment	Any instance of a running system. An MDS deployment is a running system built from the MDS frameworks.
Rate group	A group of tasks that are scheduled to execute at the same rate.
Software architecture	Software architecture introduces a level of abstraction of software with notions such as components. This abstraction facilitates the description, modeling, design, reuse, testing and validation of complex software.
Thread	An operating system entity that can perform a computation independently of other threads' computations in the system.

Framework Package 28: Time Management

Overview:

The Time Management package provides mechanisms for the initialization and management of time services.

The Time Management package includes methods for defining time frames, and transforming epochs from one time frame to another. It also includes functions for comparing, adding, and subtracting epoch and duration values as well as functions for converting time values to and from human-readable format.

Description of basic functions:

Function Name	Description
Initialize time	This function establishes the current time, establishes the database of time frames available (e.g. TAI - international atomic time) and conversions between them, and establishes the default time frame.
Get time	This function allows a caller to get the current time, in the default time frame in use.
Read time	This function converts a human-readable time value into an internal epoch value. The time frame may be specified in the human-readable value, or if not, the default is used.
Write time	This function outputs an internal epoch value to a human-readable character string. The default format of the output is: `YYYY-MM-DD:hh:mm:ss.sss<frame>`, e.g. `2002-11-06:09:51:26.321TAI`
Set time frame	This function converts an epoch value from being expressed in one time frame to being expressed in another. If the conversion is not possible, the function generates an exception.
Add epoch and duration	Adds duration to an epoch and produces a new epoch. Duration values also are expressed in a time frame, and so this operation may require a conversion of frames before doing the addition. An invalid frame conversion results in a thrown exception.
Compare epochs	This function allows the evaluation of logical expressions such as 'e1 < e2' or 'e1 >= e2', where e1 and e2 are epoch values.

Glossary:

Term	Definition

Term	Definition
Epoch	An epoch epresents a fixed point in time.
Time frame	A time frame represents a coordinate system for an epoch. A time frame is generally unique to a particular clock, though we define a few standard frames such as TAI and UTC where the assumption is made that any clock expressing time in that frame will provide some kind of synchronization mechanism to keep it accurate with respect to that frame. When dealing with time-dependent control systems that can be widely distributed over huge distances of space (and thus time), it is important to be unambiguous about the time frames in which events are measured and specified.

Framework Package 29: Unit Testing

Overview:

The Unit Testing package provides a test harness for verifying packages. It is externally provided library called CppUnit. For more information about its capabilities, refer to http://cppunit.sourceforge.net/cppunit-wiki.

Description of basic functions:

Function Name	Description
	Please refer to http://cppunit.sourceforge.net/cppunit-wiki

Framework Package 30: Value History

Overview:

The Value History package provides abstract interfaces for data containers that hold state, command, measurement, and simulation data histories.

These containers hold items like state functions accessed through state variables, measurements produced by hardware adapters, commands sent to hardware adapters, and time-stamped data produced in simulation deployments. The items may be stored in time-sorted containers. The Value History package provides insertion and retrieval interfaces on these containers.

Value History containers contain objects of user-defined types. The only thing all of these types have in common is that they are associated with a time, either a single point or a range.

As well as providing an abstract interface, the Value History package provides a few highly optimized containers that meet the abstract interfaces:

Global variable: Implements a very simple value history for a single concrete type. Because the item is stored by value it can't be abstract. This history container is appropriate for very simple histories where only the single (latest) value is ever required

Two-item Pool: Two-item history that uses a pool and atomic smart pointers to maintain a fast but thread safe container for homogeneous instances of the template state function class. As a value history this container is intended for use in situations where the control algorithm needs access to one most recent value plus one preserved value. Normally, the preserved value is the state value preserved at the start of a new constraint. The container provides a reference counted pointer index to each of the two nominal values along with a memory pool for the actual values. The pool provides space for 2 * *tsize* actual value instances. The value of *tsize* should be chosen to ensure thread safe atomic operations.

Note that this package is implemented as a sub-package within the state package.

Description of basic functions:

Function Name	Description
Update	Abstract interface for adding an item to the history.
Query	Abstract interface for accessing the history.

Glossary:

Term	Definition
Item	A data element stored in a value history.

3 Glossary Index